EZE big data pipeline

- HDFS path root: `/data/onboarding/{channel}` (channel = `online`, `branch`, `mobile`)
- Hive external table location: `/warehouse/external/onboarding`
- Lookups are stored in Hive as `lookups.branch_region_map` (branch → region, branch metadata)
- Timestamps are ISO8601 in the JSON (e.g. `2025-10-23T09:21:00Z`)
- PySpark uses SparkSession with Hive support (`enableHiveSupport()`)

1) High-level architecture (text)

1. JSON files arrive in HDFS folders every 30 minutes under:
    - `/data/onboarding/online/`
    - `/data/onboarding/branch/`
    - `/data/onboarding/mobile/`
2. Oozie Coordinator scheduled every 30 minutes:
    - detects new files (file availability)
    - triggers an Oozie workflow which runs a Spark job (PySpark) via `spark-submit`
3. PySpark job:
    - reads new JSON files for the run
    - validates mandatory fields (PAN, Aadhaar, address, nominee)
    - creates status flags (KYC_PENDING / KYC_COMPLETED / ADDRESS_MISMATCH / INVALID_DOC)
    - enriches records using Hive lookups (branch → region)
    - writes to partitioned Hive external table in ORC/Parquet (partitioned by `channel`, `onboarding_date`)

- writes processed-file manifest into HDFS for idempotence (or updates a metadata table)

4. Monitoring:

    - During processing, compute KYC elapsed time; if > 2 hours, send email alert and log to alert table

    - Oozie/CD pipeline raises alert if coordinator detects a missed file window

    - Daily summary (Hive query) emailed using Python or Oozie

2) Example JSON schema & example record

```json
{
  "customer_id": "CUST123456",
  "channel": "online",            // "online" | "branch" | "mobile"
  "onboarding_ts": "2025-10-23T09:21:00Z",
  "name": "Ramesh Kumar",
  "pan": "ABCDE1234F",
  "aadhaar": "123412341234",
  "address": {
    "line1": "12 MG Road",
    "city": "Bengaluru",
    "state": "Karnataka",
    "pincode": "560001"
  },
  "nominee": {
    "name": "Sita Kumar",
    "relation": "Spouse"
  },
  "documents": [
     {"type": "PAN", "valid": true},
     {"type": "Aadhaar", "valid": true}
  ],
  "branch_id": "BR001",           // for branch channel or to map a preferred
branch
  "kyc_completed_ts": null,       // gets populated once KYC completes
  "raw_payload": { ... }          // optional, store raw
}
```

3) PySpark processing script (main)

Save as `process_onboarding.py`. This does:

- read JSONs for a channel

- validate
- enrich via Hive lookup table
- assign statuses
- write to ORC (or Parquet) external Hive table partitioned by `channel` and `onboarding_date`
- log alerts via SMTP when KYC elapsed > 2 hours
- write processed manifest file to HDFS to prevent reprocessing

```python
# process_onboarding.py
import sys
import os
from pyspark.sql import SparkSession, functions as F, types as T
from pyspark.sql.window import Window
import smtplib
from email.mime.text import MIMEText
from datetime import datetime, timezone, timedelta
import json

# ---------- Config ----------
HIVE_DB = "onboarding_db"
LOOKUP_TABLE = "lookups.branch_region_map"  # Hive table: branch_id -> region, branch_name
OUTPUT_TABLE = "onboarding_raw"
OUTPUT_BASE_PATH = "/warehouse/external/onboarding"  # external table location root
PROCESSED_MANIFEST_DIR = "/data/onboarding/processed_manifest"
ALERT_EMAIL_SENDER = "alerts@yourdomain.com"
ALERT_EMAIL_TO = ["onboarding-team@yourdomain.com"]
SMTP_HOST = "smtp.yourdomain.com"
SMTP_PORT = 25
KYC_ALERT_THRESHOLD_HOURS = 2
# ----------------------------

def send_email(subject, body, to_addrs):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = ALERT_EMAIL_SENDER
    msg['To'] = ", ".join(to_addrs)
    s = smtplib.SMTP(SMTP_HOST, SMTP_PORT)
    s.sendmail(ALERT_EMAIL_SENDER, to_addrs, msg.as_string())
    s.quit()

def main(input_path, channel, run_id):
```

```
    spark = SparkSession.builder \
        .appName(f"process_onboarding_{channel}_{run_id}") \
        .enableHiveSupport() \
        .getOrCreate()

    # read new JSON files
    df = spark.read.json(input_path)

    # Add channel if not present or override
    df = df.withColumn("channel", F.lit(channel))

    # basic flattening
    df = df.withColumn("address_line1", F.col("address.line1")) \
           .withColumn("city", F.col("address.city")) \
           .withColumn("state", F.col("address.state")) \
           .withColumn("pincode", F.col("address.pincode"))

    # Convert onboarding timestamp to timestamp type and date partition column
    df = df.withColumn("onboarding_ts_ts", F.to_timestamp("onboarding_ts")) \
           .withColumn("onboarding_date", F.date_format("onboarding_ts_ts",
"yyyy-MM-dd"))

    # Validation UDFs / expressions
    mandatory_cols = ["pan", "aadhaar", "address_line1", "nominee"]
    # is_present checks
    df = df.withColumn("missing_pan", F.when(F.col("pan").isNull() |
(F.col("pan") == ""), True).otherwise(False)) \
           .withColumn("missing_aadhaar", F.when(F.col("aadhaar").isNull() |
(F.col("aadhaar") == ""), True).otherwise(False)) \
           .withColumn("missing_address",
F.when(F.col("address_line1").isNull() | (F.col("address_line1") == ""),
True).otherwise(False)) \
           .withColumn("missing_nominee", F.when(F.col("nominee").isNull(),
True).otherwise(False))

    # Validate docs array - simple check for existence of valid PAN/Aadhaar
docs
    df = df.withColumn("has_pan_doc", F.expr("exists(documents, x ->
x.type='PAN' and x.valid = true)")) \
           .withColumn("has_aadhaar_doc", F.expr("exists(documents, x ->
x.type='Aadhaar' and x.valid = true)"))

    # Flag invalid doc if missing or doc invalid
    df = df.withColumn("invalid_doc", (F.col("missing_pan") |
F.col("missing_aadhaar") | (~F.col("has_pan_doc")) |
(~F.col("has_aadhaar_doc"))))

    # ADDRESS_MISMATCH: example rule - pincode missing or invalid length OR
city/state blank
    df = df.withColumn("address_mismatch", (F.col("pincode").isNull() |
```

```
(F.length(F.col("pincode")) != 6) | F.col("city").isNull() |
F.col("state").isNull()))

    # KYC status: if kyc_completed_ts present => KYC_COMPLETED else
KYC_PENDING
    df = df.withColumn("kyc_completed_ts_ts",
F.to_timestamp("kyc_completed_ts")) \
           .withColumn("KYC_STATUS",
                       F.when(F.col("kyc_completed_ts_ts").isNotNull(),
F.lit("KYC_COMPLETED"))
                        .otherwise(F.lit("KYC_PENDING"))
                      )

    # Enrichment: join with lookup table in Hive
    lookup_df = spark.table(LOOKUP_TABLE)  # expected columns: branch_id,
branch_name, region
    df = df.join(lookup_df, on="branch_id", how="left")

    # Final status flags (array or columns). Keep columns for ease of querying
    df = df.withColumn("STATUS_FLAGS", F.array_distinct(
                    F.array_remove(F.array(
                        F.when(F.col("KYC_STATUS") == "KYC_PENDING",
F.lit("KYC_PENDING")).otherwise(F.lit(None)),
                        F.when(F.col("KYC_STATUS") == "KYC_COMPLETED",
F.lit("KYC_COMPLETED")).otherwise(F.lit(None)),
                        F.when(F.col("address_mismatch") == True,
F.lit("ADDRESS_MISMATCH")).otherwise(F.lit(None)),
                        F.when(F.col("invalid_doc") == True,
F.lit("INVALID_DOC")).otherwise(F.lit(None))
                    ), None)
                ))

    # Compute KYC elapsed time: If KYC_COMPLETED then diff between onboarding
and kyc_completed; else now - onboarding
    now_ts = F.current_timestamp()
    df = df.withColumn("kyc_elapsed_secs",
                       F.when(F.col("kyc_completed_ts_ts").isNotNull(),
                              F.unix_timestamp(F.col("kyc_completed_ts_ts")) -
F.unix_timestamp(F.col("onboarding_ts_ts"))
                             ).otherwise(
                              F.unix_timestamp(now_ts) -
F.unix_timestamp(F.col("onboarding_ts_ts"))
                             )
                      )

    # Raise alerts for > threshold
    threshold_secs = KYC_ALERT_THRESHOLD_HOURS * 3600
    alerts_df = df.filter(F.col("kyc_elapsed_secs") > threshold_secs)

    # Send alerts (small set) via SMTP - collect needed fields into driver
```

```
    alerts =
alerts_df.select("customer_id","branch_id","region","onboarding_ts","kyc_elaps
ed_secs","STATUS_FLAGS").limit(100).toJSON().collect()
    if alerts:
        body = "KYC processing alert: customers exceeding threshold\n\n" +
"\n".join(alerts)
        try:
            send_email(f"[ALERT] KYC delay > {KYC_ALERT_THRESHOLD_HOURS}
hours", body, ALERT_EMAIL_TO)
        except Exception as e:
            # log but don't fail the job
            print("Failed to send alert email:", e)

    # Persist to Hive external table location in partitioned layout
    # We'll write to path: /warehouse/external/onboarding/channel=
<channel>/onboarding_date=<yyyy-mm-dd>/
    out_base = OUTPUT_BASE_PATH
    write_df = df.select(
        "customer_id", "channel", "onboarding_ts", "onboarding_ts_ts",
        "onboarding_date", "name", "pan", "aadhaar", "address_line1", "city",
"state", "pincode",
        "nominee", "documents", "branch_id", "branch_name", "region",
"KYC_STATUS", "STATUS_FLAGS", "kyc_elapsed_secs", "raw_payload"
    )

    # Write as ORC/Parquet with dynamic partitioning by channel and
onboarding_date
    spark.conf.set("hive.exec.dynamic.partition","true")
    spark.conf.set("hive.exec.dynamic.partition.mode","nonstrict")

    write_df.write.mode("append") \
        .partitionBy("channel", "onboarding_date") \
        .format("parquet") \
        .option("compression","snappy") \
        .save(out_base)

    # Record processed files manifest to HDFS so we don't reprocess (optional)
    # create a manifest file path based on run_id
    manifest = {
        "run_id": run_id,
        "input_path": input_path,
        "processed_at": datetime.now(timezone.utc).isoformat(),
        "records": df.count()
    }
    manifest_path = os.path.join(PROCESSED_MANIFEST_DIR,
f"manifest_{channel}_{run_id}.json")
    # write manifest via hdfs put (use spark to write small file via
sc.hadoopFile? simpler: print to stdout and let wrapper move to HDFS)
    with open("/tmp/manifest_tmp.json","w") as f:
        f.write(json.dumps(manifest))
```

```
    # Attempt to copy to HDFS (requires hadoop client on driver)
    try:
        os.system(f"hdfs dfs -mkdir -p {PROCESSED_MANIFEST_DIR}")
        os.system(f"hdfs dfs -put -f /tmp/manifest_tmp.json {manifest_path}")
    except Exception as e:
        print("Failed to push manifest to HDFS:", e)

    spark.stop()

if __name__ == "__main__":
    # args: input_path channel run_id
    if len(sys.argv) != 4:
        print("Usage: process_onboarding.py <input_path> <channel> <run_id>")
        sys.exit(1)
    input_path = sys.argv[1]
    channel = sys.argv[2]
    run_id = sys.argv[3]
    main(input_path, channel, run_id)
```

Notes:

- The script uses `spark.read.json(input_path)` — point `input_path` to the directory or file pattern (e.g. `hdfs:///data/onboarding/online/*.json`).
- The script writes out to `OUTPUT_BASE_PATH` and uses dynamic partitions for `channel` and `onboarding_date`. You should create a Hive external table pointing to that base path (DDL below).
- The email send uses a basic SMTP conversation; replace with your mail relay or use a more robust mail client if needed.

4) Hive table DDL (external) — ORC/Parquet

Run in Hive/Beeline:

sql

```
CREATE DATABASE IF NOT EXISTS onboarding_db;

CREATE EXTERNAL TABLE IF NOT EXISTS onboarding_db.onboarding_raw (
  customer_id string,
  onboarding_ts string,
  onboarding_ts_ts timestamp,
  name string,
  pan string,
  aadhaar string,
  address_line1 string,
  city string,
  state string,
  pincode string,
  nominee struct<name:string,relation:string>,
  documents array<struct<type:string,valid:boolean>>,
  branch_id string,
  branch_name string,
  region string,
  KYC_STATUS string,
  STATUS_FLAGS array<string>,
  kyc_elapsed_secs bigint,
  raw_payload string
)
PARTITIONED BY (channel string, onboarding_date string)
STORED AS PARQUET
LOCATION '/warehouse/external/onboarding';
```

After creating the table, you may run `MSCK REPAIR TABLE onboarding_db.onboarding_raw;` to pick up existing partitions.

Optional: create a small metadata table `onboarding_db.processed_manifest` to track processed files, alerts, etc.

5) Lookup table DDL example

```sql
CREATE DATABASE IF NOT EXISTS lookups;
CREATE EXTERNAL TABLE IF NOT EXISTS lookups.branch_region_map (
  branch_id string,
  branch_name string,
  region string
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/warehouse/external/lookups/branch_region_map';
```

Populate this via load data or insert from CSV. Or create as managed table.

6) Oozie: coordinator + workflow sketch

Coordinator (runs every 30 minutes, expects files in each channel directory)

`coordinator.xml`

xml

```xml
<coordinator-app name="onboarding-coordinator" frequency="30" start="2025-10-01T00:00Z" end="2026-01-01T00:00Z" timezone="UTC"
xmlns="uri:oozie:coordinator:0.4">
  <action>
    <workflow>
      <app-path>${nameNode}/user/oozie/workflows/onboarding_workflow.xml</app-path>
      <configuration>
        <property>
          <name>input_dir</name>
          <value>/data/onboarding/${coord:nominalTimeFormat('yyyy-MM-dd-HH-mm')}/</value>
        </property>
      </configuration>
    </workflow>
  </action>

  <!-- File triggers per channel -->
  <datasets>
    <dataset name="online_files" frequency="30" timezone="UTC">
      <uri-template>hdfs://<NAMENODE_HOST>:8020/data/onboarding/online/${YEAR}-${MONTH}-${DAY}-${HOUR}-${MINUTE}/*.json</uri-template>
      <start>${coord:start}</start>
      <end>${coord:end}</end>
    </dataset>
    <dataset name="branch_files" frequency="30" timezone="UTC">
      <uri-template>hdfs://<NAMENODE_HOST>:8020/data/onboarding/branch/${YEAR}-${MONTH}-${DAY}-${HOUR}-${MINUTE}/*.json</uri-template>
      <start>${coord:start}</start>
      <end>${coord:end}</end>
    </dataset>
    <dataset name="mobile_files" frequency="30" timezone="UTC">
      <uri-template>hdfs://<NAMENODE_HOST>:8020/data/onboarding/mobile/${YEAR}-${MONTH}-${DAY}-${HOUR}-${MINUTE}/*.json</uri-template>
      <start>${coord:start}</start>
      <end>${coord:end}</end>
    </dataset>
  </datasets>

  <controls>
    <timeout>10</timeout>
    <concurrency>1</concurrency>
```

```
    <execution>FIFO</execution>
  </controls>
</coordinator-app>
```

Note: The coordinator above is a sketch. You often use Coordinator `input-events` with `uri-template` to watch for files. Some teams prefer using Oozie file system triggers or use stream processors. Replace `<NAMENODE_HOST>` and adjust start/end.

Workflow that the coordinator calls

`onboarding_workflow.xml` (a Spark action calling `process_onboarding.py`)

```xml
<workflow-app name="onboarding-workflow" xmlns="uri:oozie:workflow:0.5">
  <start to="spark-node"/>
  <action name="spark-node">
    <spark xmlns="uri:oozie:spark-action:1.0">
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <master>yarn</master>
      <mode>cluster</mode>
      <name>OnboardingProcessing_${coord:nominalTime}</name>
      <class>org.apache.spark.deploy.PythonRunner</class>
      <jar>file:///usr/hdp/current/spark-client/lib/spark-assembly.jar</jar>
      <arg>process_onboarding.py</arg>
      <arg>hdfs:///data/onboarding/${channel}/*.json</arg>
      <arg>${channel}</arg>
      <arg>${wf:id()}</arg>

<file>hdfs:///user/oozie/apps/process_onboarding/process_onboarding.py#process
_onboarding.py</file>
      <capture-output/>
    </spark>
    <ok to="end"/>
    <error to="fail"/>
  </action>
  <kill name="fail">
    <message>Workflow failed, error
message[${wf:errorMessage(wf:lastErrorNode())}]</message>
  </kill>
  <end name="end"/>
</workflow-app>
```

You'll parameterize `${channel}` for each channel and configure dataset triggers accordingly. Many implementations create three specific coordinator jobs (one per channel) to simplify templating.

7) KPI Hive queries (examples)

1. **Onboarding count by region and channel (daily)**

```sql
SELECT onboarding_date, region, channel, count(*) AS total_onboardings
FROM onboarding_db.onboarding_raw
WHERE onboarding_date = '2025-10-22'
GROUP BY onboarding_date, region, channel
ORDER BY region, channel;
```

2. **Average onboarding processing time by region/branch**

```sql
SELECT region, branch_id, AVG(kyc_elapsed_secs)/3600 AS avg_kyc_hours
FROM onboarding_db.onboarding_raw
WHERE onboarding_date BETWEEN '2025-10-01' AND '2025-10-22'
GROUP BY region, branch_id
ORDER BY avg_kyc_hours DESC;
```

3. **% of customers with incomplete documentation (INVALID_DOC)**

```sql
SELECT onboarding_date, region,
  SUM(CASE WHEN array_contains(STATUS_FLAGS, 'INVALID_DOC') THEN 1 ELSE 0 END)
AS invalid_docs,
  COUNT(*) AS total,
  (SUM(CASE WHEN array_contains(STATUS_FLAGS, 'INVALID_DOC') THEN 1 ELSE 0
END)/CAST(COUNT(*) AS DOUBLE))*100 AS pct_invalid
FROM onboarding_db.onboarding_raw
WHERE onboarding_date >= '2025-10-01'
GROUP BY onboarding_date, region
ORDER BY onboarding_date, pct_invalid DESC;
```

4. **Branches with frequent invalid entries (top 20)**

```sql
SELECT branch_id, branch_name, region,
  SUM(CASE WHEN array_contains(STATUS_FLAGS,'INVALID_DOC') THEN 1 ELSE 0 END)
AS invalid_count,
```

```
  COUNT(*) AS total
FROM onboarding_db.onboarding_raw
GROUP BY branch_id, branch_name, region
ORDER BY invalid_count DESC
LIMIT 20;
```

5. **Daily summary report SQL (for email)**

```sql
SELECT onboarding_date,
  COUNT(*) AS total_onboardings,
  SUM(CASE WHEN array_contains(STATUS_FLAGS,'INVALID_DOC') THEN 1 ELSE 0 END)
AS invalid_doc_count,
  SUM(CASE WHEN array_contains(STATUS_FLAGS,'ADDRESS_MISMATCH') THEN 1 ELSE 0
END) AS address_mismatch_count,
  SUM(CASE WHEN KYC_STATUS='KYC_COMPLETED' THEN 1 ELSE 0 END) AS kyc_completed
FROM onboarding_db.onboarding_raw
WHERE onboarding_date = date_format(current_date - interval '1' day, 'yyyy-MM-
dd')
GROUP BY onboarding_date;
```

8) Python email report script (daily)

Save as `daily_onboarding_report.py` — run once per day via Oozie or crontab.

python

```python
# daily_onboarding_report.py
import smtplib
from email.mime.text import MIMEText
import subprocess
import json
import os

SMTP_HOST = "smtp.yourdomain.com"
SMTP_PORT = 25
FROM = "reports@yourdomain.com"
TO = ["onboarding-team@yourdomain.com"]

# Example: use beeline to run SQL and capture results (hive/beeline available)
SQL = """
SELECT onboarding_date,
  COUNT(*) AS total_onboardings,
  SUM(CASE WHEN array_contains(STATUS_FLAGS,'INVALID_DOC') THEN 1 ELSE 0 END)
AS invalid_doc_count,
  SUM(CASE WHEN array_contains(STATUS_FLAGS,'ADDRESS_MISMATCH') THEN 1 ELSE 0
END) AS address_mismatch_count,
  SUM(CASE WHEN KYC_STATUS='KYC_COMPLETED' THEN 1 ELSE 0 END) AS kyc_completed
FROM onboarding_db.onboarding_raw
WHERE onboarding_date = date_format(current_date - interval '1' day, 'yyyy-MM-
dd')
GROUP BY onboarding_date;
"""

def run_beeline(sql):
    cmd = ['beeline', '-u', 'jdbc:hive2://localhost:10000/default', '-e', sql]
    p = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
text=True)
    out,err = p.communicate()
    if p.returncode != 0:
        raise Exception("Beeline failed: " + err)
    return out

def send_email(subject, body, to_addrs):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = FROM
    msg['To'] = ", ".join(to_addrs)
    s = smtplib.SMTP(SMTP_HOST, SMTP_PORT)
    s.sendmail(FROM, to_addrs, msg.as_string())
    s.quit()

if __name__ == "__main__":
    try:
        res = run_beeline(SQL)
```

```
        send_email("Daily Onboarding Summary", f"Daily Onboarding
KPI:\n\n{res}", TO)
    except Exception as e:
        # On failure, email operations team
        send_email("[ERROR] Daily Onboarding Report failed", str(e),
["ops@yourdomain.com"])
```

Run this script with Oozie or crontab daily at 06:00 AM.

9) Missed-file detection & alert (small watchdog)

If Oozie coordinator is used reliably, missed-file detection can be built into the coordinator by setting timeouts or writing a small script that checks manifest presence for the expected timeframe.

Example watchdog snippet `watchdog_check.py`:

python

```python
# watchdog_check.py
import subprocess, smtplib
from email.mime.text import MIMEText
from datetime import datetime, timedelta

HDFS_MANIFEST_DIR = "/data/onboarding/processed_manifest"
ALERT_EMAIL_TO = ["onboarding-team@yourdomain.com"]

def hdfs_ls(path):
    cmd = ["hdfs","dfs","-ls", path]
    p = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
text=True)
    out,err = p.communicate()
    if p.returncode != 0:
        return ""
    return out

def send_email(subject, body, to_addrs):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = "alerts@yourdomain.com"
    msg['To'] = ", ".join(to_addrs)
    s = smtplib.SMTP("smtp.yourdomain.com", 25)
    s.sendmail("alerts@yourdomain.com", to_addrs, msg.as_string())
    s.quit()

if __name__ == "__main__":
    # check last 30-minute window
    now = datetime.utcnow()
    window = now - timedelta(minutes=30)
    date_tag = window.strftime("%Y-%m-%d-%H-%M")  # match your manifest naming
convention
    # Use hdfs ls to search for manifest entries
    out = hdfs_ls(HDFS_MANIFEST_DIR)
    if date_tag not in out:
        send_email("[ALERT] Missing onboarding ingest", f"No manifest found
for {date_tag}. Possible missed files.", ALERT_EMAIL_TO)
```

Schedule this watchdog every 30 minutes via crontab or Oozie frequency job.

10) Deployment & testing checklist

1. **Create Hive DBs and lookup tables**, load lookup CSV for branch→region.
2. **Create Hive external table** pointing to base output path. Validate partition discovery works.

3. **Deploy PySpark script** into a shared HDFS /user/oozie/apps directory referenced by Oozie.

4. **Test locally on sample JSON**: run `spark-submit process_onboarding.py /local/path/sample_online.json online test1` (with local spark). Validate output files partitioned correctly.

5. **Test HDFS ingestion**: copy sample file into HDFS `hdfs dfs -put sample_online.json /data/onboarding/online/` and run job.

6. **Simulate KYC delays**: create synthetic samples with onboarding_ts older than 3 hours to verify alerts.

7. **Deploy Oozie coordinator + workflow** in dev cluster, validate triggers and retries.

8. **Test email delivery** for alerts and daily summary.

9. **Set up logging**: store job logs in HDFS or centralized ELK/Fluentd for easier debugging.

10. **On-call runbook**: document steps to reprocess missing files and to replay manifests.

11) Operational considerations & best practices

- **Idempotence**: keep a processed manifest (or use HDFS file move semantics) so files already processed aren't reprocessed. Use a metadata Hive table if you want more robust processing-state management.

- **Small files**: since files arrive every 30 minutes, ensure batches aren't too small. If lots of small files, consider rolling them or using Spark streaming with file stream or S3-style consolidation.

- **Schema evolution**: store raw JSON (raw_payload) for reprocessing if the schema changes.

- **Backfills & replays**: provide a replay mode that reads historical JSON and reprocesses into partitions (with overwrite per partition).

- **Testing**: unit test validation logic (e.g., PAN, Aadhaar length rules) and edge cases (nulls, malformed JSON).

- **Security**: mask or encrypt PII at rest if required (Aadhaar/PAN). Use HDFS encryption zones if mandated.

- **Monitoring**: track job durations, success/failure counts, and number of records processed in Prometheus/Grafana.

- **Retries**: Oozie workflows should be configured with retry logic. If job fails, alert and write failed-run metadata.

12) Quick troubleshooting tips

- If partitions not visible in Hive: run `MSCK REPAIR TABLE onboarding_db.onboarding_raw;`
- If email not sent: check SMTP connectivity from nodes where driver runs.
- If alerts noisy: add hysteresis (e.g., only alert once per customer unless newly exceeding threshold).
- If too many small files in HDFS: run periodic compaction into date/partition files.

🧠 What's the Goal?

You need to **build a Big Data Pipeline** for a company called **EZE** that handles **new customer onboarding data** coming from multiple sources.
This pipeline should collect, clean, validate, enrich, and analyze the data — and also send alerts or reports when something goes wrong.

Basically, you're building an **automated data system** that tells the company:

> "How fast and correctly are customers getting onboarded, and where are delays or issues happening?"

📦 Where is the data coming from?

Customer data comes from **three channels**:

1. **Online applications**
2. **Branch (in-person) registrations**
3. **Mobile app signups**

Every 30 minutes, each channel drops **JSON files** into HDFS (a big data file system).

Example folders:

```swift
/data/onboarding/online/
/data/onboarding/branch/
/data/onboarding/mobile/
```

⚙️ What the system must do

1. **Detect new files automatically**
    - When a new JSON file lands in HDFS, detect it within a few minutes.
    - You can use **Oozie** (a scheduler) or **Spark Streaming** to do this.
2. **Validate the data**
    - Check that required information is present:
        - PAN number
        - Aadhaar number
        - Address
        - Nominee details
    - If any of these are missing → mark as **invalid**.
3. **Enrich the data**
    - Add extra info like **branch name** and **region** using a **lookup table** stored in Hive.
4. **Transform and Store**
    - Use **PySpark** to process and clean the data.
    - Write the cleaned data to **Hive external tables**.
    - Tables should be partitioned by:
        - `channel` (online, branch, mobile)
        - `onboarding_date`
5. **Generate Status Flags**
   For each record, create flags such as:
    - `KYC_PENDING`
    - `KYC_COMPLETED`
    - `ADDRESS_MISMATCH`

- `INVALID_DOC`

6. **Analyze the data**
   - Run Hive queries to answer questions like:
     - How long does onboarding take per branch or region?
     - What % of customers have incomplete documents?
     - Which branches have frequent data issues?
7. **Send Alerts**
   - If any customer's KYC takes more than **2 hours**, send an **email alert**.
   - If a file isn't processed within **30 minutes**, also send an alert.
8. **Send Daily Summary Reports**
   - Every day, automatically email a summary report (e.g., total onboardings, issues, etc.) to the onboarding team.

🧩 Tools You'll Use

Task	Tool
Data Storage	**HDFS**
Data Processing	**Apache Spark (PySpark)**
Data Querying	**Hive (with Parquet/ORC format)**
Scheduling / Automation	**Oozie**
Dashboards	**Tableau / Power BI / Superset (optional)**
Alerting	**Python email script or Oozie email action**

🚀 How You'll Build It (step by step)

1. **Data Ingestion**
   - Simulate three data sources producing JSON files.

    - Store them in HDFS folders (one per channel).
2. **Data Validation & Transformation**
    - Write a **PySpark script** that:
        - Reads JSON files from HDFS
        - Validates mandatory fields
        - Adds region/branch info
        - Assigns KYC status flags
        - Writes cleaned output to Hive tables
3. **Data Storage**
    - Create Hive external tables partitioned by `channel` and `date`.
4. **Scheduling**
    - Use **Oozie** to run the PySpark job automatically every 30 minutes.
5. **Analytics**
    - Write Hive queries to generate onboarding KPIs and insights.
6. **Alerts**
    - Add a Python script to check for:
        - Customers with delayed KYC (> 2 hours)
        - Missing/unprocessed files
    - Send alerts via email automatically.
7. **Reporting**
    - Another daily script runs Hive queries and emails a summary to the team.

📊 What You'll End Up With

By the end, you'll have:

- ✅ Automated data ingestion and validation pipeline
- ✅ Clean, enriched onboarding data stored in Hive
- ✅ KYC performance dashboards and reports

- ✅ Automatic email alerts for delays or errors
- ✅ Daily summary reports for the team

🧭 In short:

> You’re creating a **fully automated big data system** that continuously monitors customer onboarding performance, identifies bottlenecks, and keeps the team informed in real-time.